CMSC 631 — Program Analysis and Understanding
Fall 2007

Data Flow Analysis

Compiler Structure

- Source code parsed to produce AST
- AST transformed to CFG
- Data flow analysis operates on control flow graph (and other intermediate representations)
Control-Flow Graph (CFG)

- A directed graph where
  - Each node represents a statement
  - Edges represent control flow

- Statements may be
  - Assignments $x := y \text{ op } z$ or $x := \text{ op } z$
  - Copy statements $x := y$
  - Branches \texttt{goto L} or if $x \text{ relop } y$ goto L
  - etc.

Control-Flow Graph Example

```plaintext
x := a + b;
y := a * b;
while (y > a) {
  a := a + 1;
x := a + b
}
```

![Control-Flow Graph Example Diagram]
Variations on CFGs

• Usually don’t include declarations (e.g., int x;)
  • But there’s usually something in the implementation

• Useful to have a unique entry and exit node
  • Treat them differently during dataflow analysis

• May group statements into basic blocks
  • A sequence of instructions with no branches into or out of the block

Control-Flow Graph w/Basic Blocks

x := a + b;
y := a * b;
while (y > a + b) {
  a := a + 1;
x := a + b
}

• Can lead to more efficient implementations
• But more complicated to explain, so...
  • We’ll use single-statement blocks in lecture today
Graph Example with Entry and Exit

```plaintext
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
x := a + b
}

• All nodes without a (normal) predecessor should be pointed to by entry
• All nodes without a successor should point to exit
```

CFG vs. AST

• CFGs are much simpler than ASTs
  - Fewer forms, less redundancy, simple expressions

• But...AST is a more faithful representation
  - CFGs introduce temporaries
  - Lose block structure of program

• So for AST,
  - Easier to report error + other messages
  - Easier to explain to programmer
  - Easier to unparse to produce readable code
Data Flow Analysis

- A framework for proving facts about programs
- Reasons about lots of little facts
- Little or no interaction between facts
  - Works best on properties about how program computes
- Based on all paths through program
  - Including infeasible paths

Available Expressions

- Expression \( e \) is available at program point \( p \) if
  - \( e \) is computed on every path to \( p \), and
  - the value of \( e \) has not changed since the last time \( e \) is computed on \( p \)

- Optimization
  - If an expression is available, need not be recomputed
    - (At least, if it’s still in a register somewhere)
Data Flow Facts

• Is expression $e$ available?
• Facts:
  ■ $a + b$ is available
  ■ $a \cdot b$ is available
  ■ $a + 1$ is available

Gen and Kill

• What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := a + b$</td>
<td>$a + b$</td>
<td></td>
</tr>
<tr>
<td>$y := a \cdot b$</td>
<td>$a \cdot b$</td>
<td></td>
</tr>
<tr>
<td>$a := a + 1$</td>
<td>$a + 1$, $a + b$, $a \cdot b$</td>
<td></td>
</tr>
</tbody>
</table>
Computing Available Expressions

Terminology

- A *join point* is a program point where two branches meet.

- Available expressions is a *forward must* problem:
  - Forward = Data flow from *in* to *out*
  - Must = At join point, property must hold on all paths that are joined
Data Flow Equations

• Let s be a statement
  ▪ succ(s) = { immediate successor statements of s }
  ▪ pred(s) = { immediate predecessor statements of s }
  ▪ In(s) = program point just before executing s
  ▪ Out(s) = program point just after executing s

  • In(s) = \( \bigcap_{s' \in \text{pred}(s)} \text{Out}(s') \)
  • Out(s) = Gen(s) \( \bigcup (\text{In}(s) - \text{Kill}(s)) \)
    ▪ These are also called transfer functions

Liveness Analysis

• A variable v is live at program point p if
  ▪ v will be used on some execution path originating from p...
    ▪ before v is overwritten

• Optimization
  ▪ If a variable is not live, no need to keep it in a register
  ▪ If variable is dead at assignment, can eliminate assignment
**Data Flow Equations**

- Available expressions is a forward must analysis
  - Data flow propagate in same dir as CFG edges
  - Expr is available only if available on all paths

- Liveness is a backward may problem
  - To know if variable live, need to look at future uses
  - Variable is live if used on some path

- \( \text{Out}(s) = \bigcup_{s' \in \text{succ}(s)} \text{In}(s') \)
- \( \text{In}(s) = \text{Gen}(s) \cup (\text{Out}(s) - \text{Kill}(s)) \)

**Gen and Kill**

- What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := a + b )</td>
<td>( a, b )</td>
<td>( x )</td>
</tr>
<tr>
<td>( y := a \times b )</td>
<td>( a, b )</td>
<td>( y )</td>
</tr>
<tr>
<td>( y &gt; a )</td>
<td>( a, y )</td>
<td></td>
</tr>
<tr>
<td>( a := a + 1 )</td>
<td>( a )</td>
<td>( a )</td>
</tr>
</tbody>
</table>
Computing Live Variables

\[ \{a, b\} \rightarrow x := a + b \]
\[ \{x, a, b\} \rightarrow y := a \times b \]
\[ \{x, y, a, b\} \rightarrow y > a \]
\[ \{y, a, b\} \rightarrow a := a + 1 \]
\[ \{y, a, b\} \rightarrow x := a + b \]

Very Busy Expressions

• An expression \( e \) is very busy at point \( p \) if
  - On every path from \( p \), expression \( e \) is used before any component of \( e \) is changed

• Optimization
  - Can hoist very busy expression computation to \( p \)

• What kind of problem?
  - Forward or backward?  backward
  - May or must?  must
Reaching Definitions

• A definition of a variable \( v \) is an assignment to \( v \)
• A definition of variable \( v \) reaches point \( p \) if
  ■ There is no intervening assignment to \( v \)

• Also called def-use information

• What kind of problem?
  ■ Forward or backward? forward
  ■ May or must? may

Space of Data Flow Analyses

<table>
<thead>
<tr>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reaching definitions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live variables</td>
</tr>
</tbody>
</table>

• Most data flow analyses can be classified this way
  ■ A few don’t fit: bidirectional analysis
• Lots of literature on data flow analysis
Data Flow Facts and Lattices

- Typically, data flow facts form a lattice
  - Example: Available expressions

```
  a+b, a*b, a+1
  a+b, a*b
  a*b, a+1
  a+b
  (none)
```

```
\text{“top”}
```

```
\text{“bottom”}
```

Partial Orders

- A partial order is a pair \((P, \leq)\) such that
  - \(\leq \subseteq P \times P\)
  - \(\leq\) is reflexive: \(x \leq x\)
  - \(\leq\) is anti-symmetric: \(x \leq y\) and \(y \leq x \Rightarrow x = y\)
  - \(\leq\) is transitive: \(x \leq y\) and \(y \leq z \Rightarrow x \leq z\)
Meet and Join Operations

- \( \sqcap \) is the **meet** or greatest lower bound operation:
  - \( x \sqcap y \leq x \) and \( x \sqcap y \leq y \)
  - if \( z \leq x \) and \( z \leq y \), then \( z \leq x \sqcap y \)

- \( \sqcup \) is the **join** or least upper bound operation:
  - \( x \leq x \sqcup y \) and \( y \leq x \sqcup y \)
  - if \( x \leq z \) and \( y \leq z \), then \( x \sqcup y \leq z \)

Lattices

- A partial order \((P, \leq)\) is a **lattice** if meet and join exist for every pair of elements in \( P \)
- A lattice has unique elements \( \bot \) and \( \top \) such that
  - \( x \sqcap \bot = \bot \) \quad \( x \sqcup \bot = x \)
  - \( x \sqcap \top = x \) \quad \( x \sqcup \top = \top \)

- In a lattice, \( x \leq y \) iff \( x \sqcap y = x \)
- In a lattice, \( x \leq y \) iff \( x \sqcup y = y \)

- A partial order is a **complete lattice** if meet and join are defined on any set \( S \subseteq P \)
Useful Lattices

- \((2^S, \subseteq)\) forms a lattice for any set \(S\)
  - \(2^S\) is the powerset of \(S\) (set of all subsets)

- If \((S, \leq)\) is a lattice, so is \((S, \geq)\)
  - I.e., lattices can be flipped

- The lattice for constant propagation

Forward Must Data Flow Algorithm

\[ \text{Out}(s) = \text{Top} \text{ for all statements } s \]

- // Slight acceleration: Could set \(\text{Out}(s) = \text{Gen}(s) \cup (\text{Top} - \text{Kill}(s))\)

\[ W := \{ \text{all statements} \} \quad \text{(worklist)} \]

repeat
  Take \(s\) from \(W\)
  \[ \text{In}(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s') \]
  \[ \text{temp} := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s)) \]
  if (temp \(!=\) Out(s)) {
    Out(s) := temp
    W := W \cup \text{succ}(s)
  }
until \(W = \emptyset\)
Monotonicity

- A function \( f \) on a partial order is monotonic if
  \[
  x \leq y \Rightarrow f(x) \leq f(y)
  \]

- Easy to check that operations to compute \( \text{In} \) and \( \text{Out} \) are monotonic
  - \( \text{In}(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s') \)
  - \( \text{Out}(s) := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s)) \)

- Putting these two together
  - \( \text{Out}(s) := f_s(\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')) \)

Termination

- We know the algorithm terminates because
  - The lattice has finite height
  - The operations to compute \( \text{In} \) and \( \text{Out} \) are monotonic
  - On every iteration, we remove a statement from the worklist and/or move down the lattice
Forward Data Flow, Again

Out(s) = Top \hspace{1em} \text{for all statements } s

W := \{ \text{all statements} \} \hspace{1em} \text{(worklist)}

repeat
    Take s from W
    temp := f (\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')) \hspace{1em} (f \text{ monotonic transfer fn})
    if (temp \neq \text{Out}(s)) {
        Out(s) := temp
        W := W \cup \text{succ}(s)
    }
until W = \emptyset

Lattices \((P, \leq)\)

• Available expressions
  • \(P = \) sets of expressions
  • \(S1 \cap S2 = S1 \cap S2\)
  • Top = set of all expressions

• Reaching Definitions
  • \(P = \) set of definitions (assignment statements)
  • \(S1 \cap S2 = S1 \cup S2\)
  • Top = empty set
Fixpoints

- We always start with Top
  - Most optimistic assumption
    - “every expression is available,” “no defns reach this point”
  - Strongest possible hypothesis
    - = true of fewest number of states
- Revise as we encounter contradictions
  - Always move down in the lattice (with meet)
- Result: A greatest fixpoint

Lattices (P, ≤), cont’d

- Live variables
  - P = sets of variables
  - S1 ∩ S2 = S1 ∪ S2
  - Top = empty set
- Very busy expressions
  - P = set of expressions
  - S1 ∩ S2 = S1 ∩ S2
  - Top = set of all expressions
Forward vs. Backward

\[ \text{Out}(s) = \text{Top} \text{ for all } s \]
\[ W := \{ \text{all statements} \} \]
repeat
  Take s from W
  \begin{align*}
  \text{temp} &:= f(\bigcap \text{pred}(s) \setminus \text{Out}(s')) \\
  \text{if (temp} &!\text{= Out}(s)) \{ \\
  \text{Out}(s) &:= \text{temp} \\
  W &:= W \cup \text{succ}(s) \\
  \}
  \end{align*}
until \( W = \emptyset \)

\[ \text{In}(s) = \text{Top} \text{ for all } s \]
\[ W := \{ \text{all statements} \} \]
repeat
  Take s from W
  \begin{align*}
  \text{temp} &:= f(\bigcap \text{successors}(s) \setminus \text{In}(s')) \\
  \text{if (temp} &!\text{= In}(s)) \{ \\
  \text{In}(s) &:= \text{temp} \\
  W &:= W \cup \text{pred}(s) \\
  \}
  \end{align*}
until \( W = \emptyset \)

Termination Revisited

• How many times can we apply this step:
  - temp := \( f(\bigcap s \setminus \text{pred}(s)) \text{Out}(s') \)
    if (temp != Out(s)) \{ ... \}

• Claim: Out(s) only “shrinks”
  - Proof: Out(s) starts out as top
  - So temp \( \leq \text{Top} \) after first step
  - Assume Out(s') shrinks for all predecessors s' of s
  - Then \( \bigcap s' \in \text{pred}(s) \text{Out}(s') \) shrinks
  - Since \( f \) monotonic, \( f(\bigcap s \setminus \text{pred}(s) \text{Out}(s')) \) shrinks
Termination Revisited (cont’d)

• A descending chain in a lattice is a sequence
  - $x_0 > x_1 > x_2 > \ldots$

• The height of a lattice is the length of the longest descending chain in the lattice

• Then, dataflow must terminate in $O(nk)$ time
  - $n = \#$ of statements in program
  - $k = \text{height of lattice}$
  - assumes meet operation takes $O(1)$ time

Least vs. Greatest Fixpoints

• Dataflow tradition: Start with Top, use meet
  - To do this, we need a complete meet semilattice with top, of finite height
    - complete meet semilattice = meets defined for any set
    - finite height ensures termination
  - Computes greatest fixpoint

• Denotational semantics tradition: Start with Bottom, use join
  - Computes least fixpoint
Distributive Data Flow Problems

• By monotonicity, we also have
  \[ f(x \sqcap y) \leq f(x) \sqcap f(y) \]

• A function \( f \) is distributive if
  \[ f(x \sqcap y) = f(x) \sqcap f(y) \]

Benefit of Distributivity

• Joins lose no information

\[
k(h(f(T) \sqcap g(T))) = \\
k(h(f(T)) \sqcap h(g(T))) = \\
k(h(f(T))) \sqcap k(h(g(T)))
\]
Accuracy of Data Flow Analysis

• Ideally, we would like to compute the meet over all paths (MOP) solution:
  • Let \( f_s \) be the transfer function for statement \( s \)
  • If \( p \) is a path \( \{s_1, ..., s_n\} \), let \( f_p = f_n;...;f_1 \)
  • Let \( \text{path}(s) \) be the set of paths from the entry to \( s \)

\[
\text{MOP}(s) = \bigcap_{p \in \text{path}(s)} f_p(T)
\]

• If a data flow problem is distributive, then solving the data flow equations in the standard way yields the MOP solution

What Problems are Distributive?

• Analyses of how the program computes
  • Live variables
  • Available expressions
  • Reaching definitions
  • Very busy expressions

• All Gen/Kill problems are distributive
A Non-Distributive Example

- Constant propagation

```
x := 1
\arrow{y := 2}
\arrow{z := x + y}
\arrow{x := 2}
\arrow{y := 1}
```

- In general, analysis of what the program computes in not distributive

Practical Implementation

- Data flow facts = assertions that are true or false at a program point

- Represent set of facts as bit vector
  - $\text{Fact}_i$ represented by bit $i$
  - Intersection = bitwise and, union = bitwise or, etc

- “Only” a constant factor speedup
  - But very useful in practice
**Basic Blocks**

- A *basic block* is a sequence of statements s.t.
  - No statement except the last in a branch
  - There are no branches to any statement in the block except the first

- In practical data flow implementations,
  - Compute Gen/Kill for each basic block
    - Compose transfer functions
  - Store only In/Out for each basic block
  - Typical basic block ~5 statements

**Order Matters**

- Assume forward data flow problem
  - Let \( G = (V, E) \) be the CFG
  - Let \( k \) be the height of the lattice

- If \( G \) acyclic, visit in topological order
  - Visit head before tail of edge

- Running time \( O(|E|) \)
  - No matter what size the lattice
Order Matters — Cycles

- If $G$ has cycles, visit in reverse postorder
  - Order from depth-first search
- Let $Q = \text{max } \# \text{ back edges on cycle-free path}$
  - Nesting depth
  - Back edge is from node to ancestor on DFS tree
- Then if $\forall x. f(x) \leq x$ (sufficient, but not necessary)
  - Running time is $O((Q + 1)|E|)$

Flow-Sensitivity

- Data flow analysis is flow-sensitive
  - The order of statements is taken into account
  - I.e., we keep track of facts per program point
- Alternative: Flow-insensitive analysis
  - Analysis the same regardless of statement order
  - Standard example: types
    
    ```
    {-!} x : int / x := ... {-!} x : int /
    ```
**Terminology Review**

- Must vs. May
  - (Not always followed in literature)
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Distributive vs. Non-distributive

**Data Flow Analysis and Functions**

- What happens at a function call?
  - Lots of solutions in data flow analysis literature
- In practice, only analyze one procedure at a time
- Consequences
  - Call to function kills all data flow facts
  - May be able to improve depending on language, e.g., function call may not affect locals
More Terminology

- An analysis that models only a single function at a time is *intraprocedural*
- An analysis that takes multiple functions into account is *interprocedural*
- An analysis that takes the whole program into account is...guess?

- Note: *global* analysis means “more than one basic block,” but still within a function

Data Flow Analysis and The Heap

- Data Flow is good at analyzing local variables
  - But what about values stored in the heap?
  - Not modeled in traditional data flow

- In practice: \( *x := e \)
  - Assume all data flow facts killed (!)
  - Or, assume write through \( x \) may affect any variable whose address has been taken

- In general, hard to analyze pointers
Data Flow Analysis and Optimization

• Moore’s Law: Hardware advances double computing power every 18 months.

• Proebsting’s Law: Compiler advances double computing power every 18 years.

• We’ll focus on other uses of data flow analysis in this class (later in the semester)